IMPROVING THE EFFICIENCY OF A NATURAL LANGUAGE PROCESSOR

Alfred H. Mossler

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

IMPROVING THE EFFICIENCY OF A NATURAL LANGUAGE PROCESSOR

by

Alfred H. Mossler

Thesis Advisor: G. E. Heidorn

June 1972

Approved for public release; distribution unlimited.

Improving the Efficiency of a Natural Language Processor

by

Alfred H. Mossler
Captain, United States Marine Corps
B.S., North Carolina State University, 1966

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1972

ABSTRACT

NLP is a processor being developed at the Naval Postgraduate School for research in natural language man-machine communication. With this system text can be translated into an entity-attribute-value information structure, and such a structure can be translated into text. These two processes, called decoding and encoding, respectively, are specified by writing "rules" in a language designed for this system.

This thesis reports on a scheme for storing these rules in the computer in a compact fashion, and describes the related routines. The savings in core storage and CPU time achieved by using this scheme are given for a particular application of NLP.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

The increased use of computers to solve problems is due to the greater availability of computing machinery and the associated increase in reliability, speed and accuracy. One dominant factor which seems to discourage some users from computer applications is that man-machine interaction is normally accomplished through a programming language. Thus, there is a requirement for familiarity with such a language or, as more often is the situation, for having programming personnel perform the interaction for the user. The second method mentioned is rapidly becoming intolerable, for often a communication gap develops between the programmer and the user. Also, the cost of programming personnel appears to be increasing as rapidly as other computer operating costs are decreasing, and this trend is expected to continue in the near future.

A solution to the spiraling cost problem is to automate the programming function. One possibility for doing this is to have the computer become a natural language processor. Such a processor could accept natural language statements and questions as input, utilize syntactic and semantic information to translate the input into an internal data structure, and then from this produce a computer program to solve the stated problem. The difficulty in creating such a processor is that most natural languages are ambiguous and

imprecise in their structure and are not readily adaptable for computer application. But, within the last few years the fields of artificial intelligence and linguisitics have met with some success in formally describing natural languages such as English. Some examples are Noam Chomsky's Theory of Tranformational Grammar [1] and Sydney Lamb's Stratificational Grammar [2].

A. A NATURAL LANGUAGE PROCESSOR

Currently there is a research project at the Naval Postgraduate School on a natural language processor called NLP [3,4]. This system provides a rule language and associated processors for "decoding" natural language text into an Internal Problem Description (IPD) and for "encoding" an IPD into text in some natural language or some programming language.

The current application of NLP is one for producing GPSS simulation programs for simple queuing problems. This application of NLP is referred to as NLPQ. The objective of NLPQ is to enable an analyst to solve simple queuing problems by describing the problem to the computer in English and receiving as output from NLPQ a GPSS program.

Work on NLPQ has been reported in a number of masters theses. Reference 5 describes an Internal Problem Description (IPD) for storing simple queuing problems and a procedure which encodes the IPD into a GPSS program. Reference 6 extended the GPSS encoding procedures and provided additional encoding procedures which translate the IPD into an equivalent

English description of the queuing problem. Reference 7 supplemented NLPQ with an interactive question answer scheme for generating the IPD. Reference 8 added an interrogator for inspecting the IPD to insure that a proper GPSS program will be produced.

The programming language used for NLP is FORTRAN IV, and the program runs under the CP/CMS time sharing system on an IBM 360/67 computer.

B. THESIS OBJECTIVE

The research objective of this thesis was to develop a scheme for storing the compiled decoding and encoding rules of NLP in a more compact form. A secondary goal was to do this in such a way as to reduce the amount of "paging" performed by CP/CMS and thereby reduce the CPU time required to execute NLP.

C. ORGANIZATION OF THESIS

Section II of this report describes the MAIN routine of NLP, some of the available NLP commands, parameters, switches and some printing commands available to the user. Section III reports on the compilation of NLP rules and the processing of named record definitions. Section IV describes the decoding and encoding processes and the operation of the NLP interpreter (CRSEG). Section V discusses the savings in core storage and CPU time achieved, and section VI contains the conclusions and some recommendations for future NLP research.

In order to understand this thesis a familiarity with the material in Refs. 3 and 4 is necessary. Listings of the

FORTRAN program are available from Professor George E. Heidorn at the Naval Postgraduate School.

II. MAIN NLP PROGRAM

Before NLP can process input text, it must compile the rules and named record definitions which specify a particular NLP application. The information obtained by performing this function is stored in the CELL array and in the A-array. These arrays and the information they contain are referred to as the Information Storage Structure (ISS) in this thesis. The ISS does not include information obtained while processing input text (i.e. the IPD). The CELL array contains the named records and segment type records, while the A-array contains the compiled rules. The CELL array is described in Ref. 4. The A-array will be discussed in detail in the next two chapters.

This section will discuss in general terms how the NLP program initially sets-up the ISS, and also the functions of some of the parameters and switches which the user can set. Because of the importance of being able to look at information actually stored in the ISS, a discussion of print commands is also presented.

A. THE MAIN ROUTINE

The NLP program has five basic sections, named NLP, PRNAMS, DECODE, ENCODE and LPR. The main program which starts the system is in the NLP section and is referred to as MAIN routine or NLP MAIN. A flow chart of the MAIN routine execution is

shown in Figure 1. The function of NLP MAIN is to initialize variables, the CELL array and the A-array, process NLP commands, and store the ISS in an output file.

Initialization is accomplished either internally or from a previously written file. An example of how the user interacts with the system during initialization and transferring of the ISS to an output file is included in Ref. 3. Once initialization is completed the system is ready to accept NLP commands. A list of commands and their function is presented in Appendix A. A command must begin in column 1 and end with a colon. The specific command determines which routine NLP MAIN will transfer control to.

B. PARAMETERS AND SWITCHES

NLP has a number of parameters and switches which can be set in a NAMELIST statement to control program execution. Switches are variables which can have values of "true" or "false", and parameters are variables which can take on other values. A listing and description of parameters and switches is contained in Appendix B and Appendix C respectively. The purpose of these variables is to allow the user to alter program execution to a small degree, and to obtain tracings of program execution. Parameter and switch variables can be set whenever the program requests optional data. A sample reply to such a request is:

```
&p prtsw=t, out6=8 &end
```

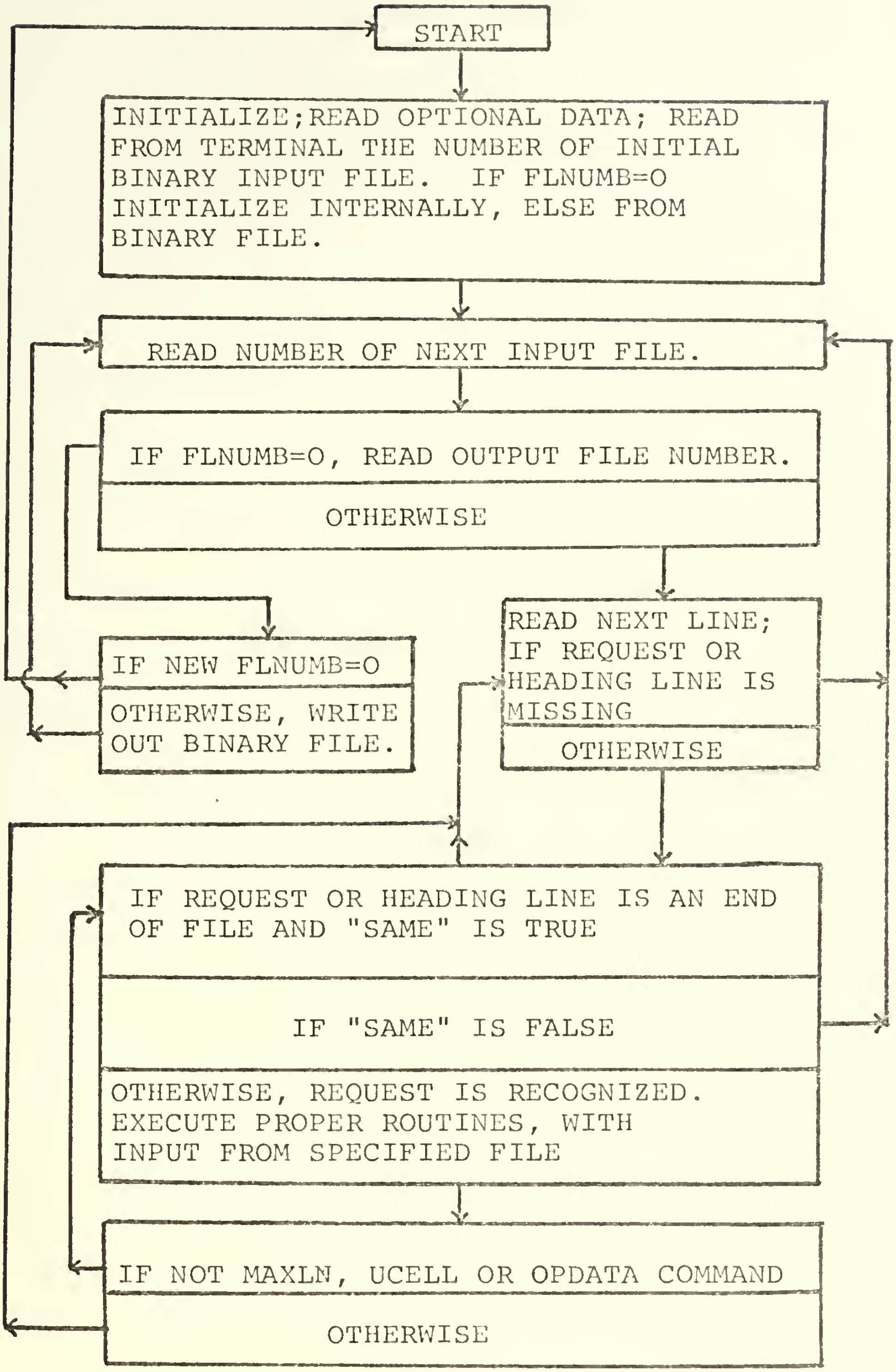

START
INITIALIZE;READ OPTIONAL DATA; READ FROM TERMINAL THE NUMBER OF INITIAL BINARY INPUT FILE. IF FLNUMB=O INITIALIZE INTERNALLY, ELSE FROM BINARY FILE.
READ NUMBER OF NEXT INPUT FILE.
IF FLNUMB=O, READ OUTPUT FILE NUMBER.
OTHERWISE
IF NEW FLNUMB=O
OTHERWISE, WRITE OUT BINARY FILE.
READ NEXT LINE; IF REQUEST OR HEADING LINE IS MISSING
OTHERWISE
IF REQUEST OR HEADING LINE IS AN END OF FILE AND "SAME" IS TRUE
IF "SAME" IS FALSE
OTHERWISE, REQUEST IS RECOGNIZED. EXECUTE PROPER ROUTINES, WITH INPUT FROM SPECIFIED FILE
IF NOT MAXLN, UCELL OR OPDATA COMMAND
OTHERWISE

Figure 1. NLP MAIN Processing

The characters "&p" must be preceeded by a blank space. The example shown causes the print switch to be set and designates file 8 as the output file for any write statements having OUT6 as their output file parameter. Certain parts of the NLP program allow only certain parameters and switches to be set. The program listing must be consulted to determine when such specific variables can be set. Tracings of program execution are especially helpful in "debugging" revisions and additions to NLP.

C. PRINT COMMAND

The print command can be used to print selected information stored in the ISS (Information Storage Structure). Print command format and examples are

```
COMMANDS:
PRINT          SEgmentype [name],[level-number]:
PRINT            REcord [number],[level-number]:
PRINT           INdicator [name],[level-number]:
PRINT           ATtribute [name],[level-number]:
PRINT             ROutine [name],[level-number]:
PRINT             *       [name],[level-number]:
PRINT                   MEmory,[level-number]:
PRINT       ['named-record-name'],[level-number]:
PRINT             ARray [number]:
PRINT             ARray [number]-[number]:
```

EXAMPLES:

```
PRINT  SE  VERB:
PRINT  ARRAY  1 - 200:
PRINT AR 5z:
PRINT 'ACTNLIST',2:
```

When specifying the type of information to be printed, only the first two characters need be entered after the print command. Hexadecimal numbers are indicated by "z" following the number. The level-number is optional and specifies to what level of detail the output should be. For the last example above, the named record for 'ACTNLIST' and any records pointed to by 'ACTNLIST' are printed.

Printing of A-array information is a feature which was implemented as part of the work done for this thesis. A detailed discussion of the A-array content is presented in the next section, where examples will show what information is stored in the A-array.

III. COMPILING OF RULES AND NAMED RECORD DEFINITIONS

The rules and named record definitions are the elements which specify a unique NLP application. An example of such an application is NLPQ. The NLP program is language-independent, but a specific set of rules determines how NLP will process input and output. Thus, the first function of NLP is to compile and process rules.

Reference 4 describes in detail the rule and named definitions of NLP. There are basically two types of rules, decoding and encoding. An example of a decoding rule is:

```
    VERBS(ED) E D  -->  VERBP(SUP(VERBS),PASTPART,PASTF)
```

An example of an encoding rule is:

```
        VERBP(PASTPART)  -->  VERBS(SUP(VERBP)) E D
```

Each rule has a "left" part and a "right" part, separated by an "arrow". The purpose of a rule is to specify the grammar and conditions under which segment types on the right-part will be created after all the conditions on the left-part are satisfied.

Named record definitions provide information about words and concepts which the system can recognize and process. An example of a named record definition is:

```
            WAIT  ('ACTIVITY',NSFX,S,ING,ED,ER)
```

Rules and named record definitions are compiled and processed by the PRNAMS section of NLP. Within PRNAMS are the routines PRULES, PRNREC, PRCNLB, GETSYM and CODE. PRULES processes the rules, and PRNREC processes the named record definitions. Both make use of PRCNLB, GETSYM and CODE. Rules are stored in the A-array and the named records are stored in the CELL array. The segment type records produced during rule processing are stored in the CELL array.

Familiarity with the concept of segment type records is important for an understanding of rule processing. Thus, this section begins with a brief discussion of segment type records. The compiling process of a rule will be described by explaining what effects PRULES, PRCNLB, GETSYM and CODE have on a rule. The processing of named record definitions is also described because the same routines which process the rules, except for PRULES, also process the attribute specifications of named record definitions.

A. SEGMENT TYPE RECORDS

The names of the elements on the left and right parts of a rule are called segment types. The segment types on the left side of a rule usually contain conditions in parentheses, and when the input to the rules satisfies all the conditions on the left then segment records are created according to the segment types and their related creation specifications on the right side. The creation specifications are in parentheses. Segment type records are stored in the CELL array

in an entity-attribute-value fashion as described in Reference 4. Appendix D lists the attributes of segment type records.

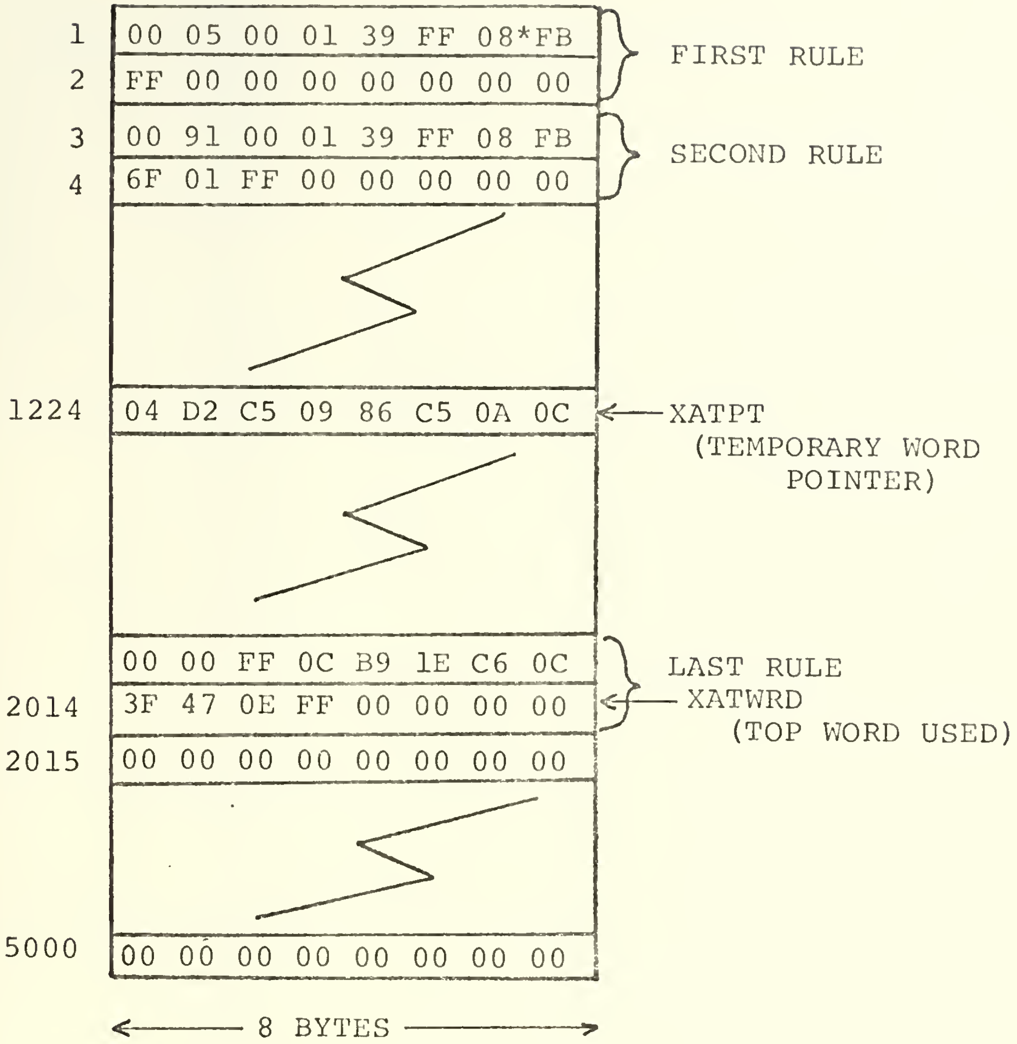
B. PROCESSING OF RULES (PRULES)

PRULES is the routine that processes the decoding and encoding rules. Basically, PRULES has three parts, which are initialization, a left-part processor, and a right-part processor. The input to PRULES is a set of rules to be compiled and stored in the A-array.

1. The A-array Structure

The physical structure of the A-array is shown in Figure 2, and the conceptual structure is shown in Figure 3, for a sample rule. The A-array stores the compiled decoding and encoding rules for a particular NLP application. These rules do not change during execution of the NLP program and thus can be compiled and stored as compactly as possible, keeping in mind economic retrieval of the A-array contents. The A-array is a REAL*8 one-dimensional array of 5000 elements. This particular physical structure was chosen because the largest IBM 360 fortran variable is REAL*8 (64 BITS), and the addressing limitation of INTEGER*2 subscripts is 32,768. All addresses in NLP require at most two bytes (because both the CELL array and the A-array have less than 32,768 elements). The information in the A-array is stored in byte format using eight bytes per element. This provides for maximum storage capacity of approximately 256k bytes.

A-ARRAY



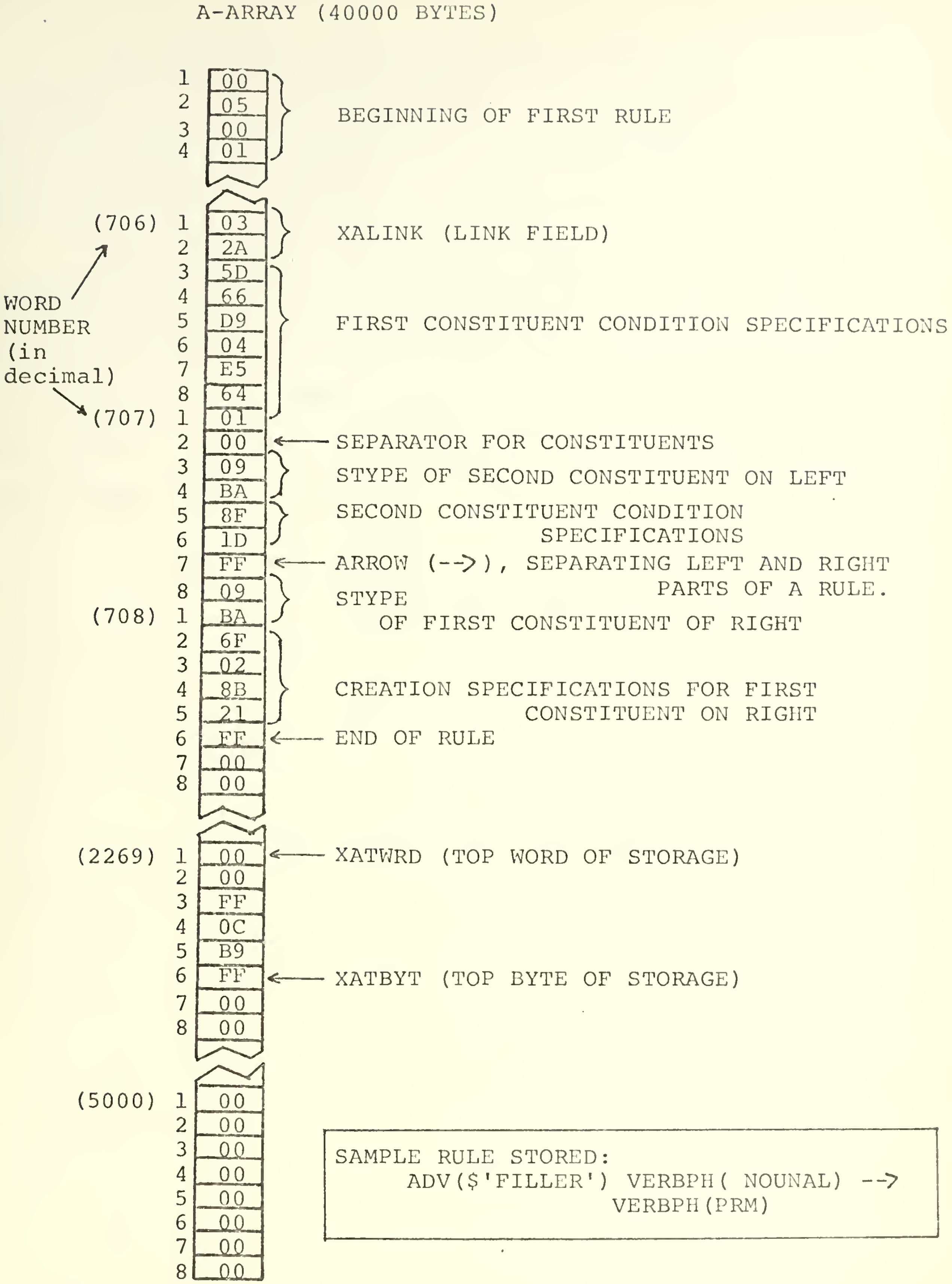
```
1     00 05 00 01 39 FF 08*FB   }
2     FF 00 00 00 00 00 00 00   }  FIRST RULE
3     00 91 00 01 39 FF 08 FB   }
4     6F 01 FF 00 00 00 00 00   }  SECOND RULE

1224  04 D2 C5 09 86 C5 0A 0C   <--XATPT
                                     (TEMPORARY WORD
                                         POINTER)

      00 00 FF 0C B9 1E C6 0C   }  LAST RULE
2014  3F 47 0E FF 00 00 00 00   }<-- XATWRD
                                       (TOP WORD USED)
2015  00 00 00 00 00 00 00 00

5000  00 00 00 00 00 00 00 00

      <-------- 8 BYTES -------->
```

REAL*8   (64 BITS OR 8 BYTES)

DIMENSION (5000)

* ENTRIES ARE HEXADECIMAL NUMBERS

Figure 2. Physical Structure and Storage of the A-Array


A-ARRAY (40000 BYTES)
1 00
2 05
3 00
4 01
BEGINNING OF FIRST RULE
(706) 1 03
2 2A
XALINK (LINK FIELD)
3 5D
4 66
5 D9
6 04
7 E5
8 64
FIRST CONSTITUENT CONDITION SPECIFICATIONS
WORD NUMBER (in decimal)
(707) 1 01
2 00
SEPARATOR FOR CONSTITUENTS
3 09
4 BA
STYPE OF SECOND CONSTITUENT ON LEFT
5 8F
6 1D
SECOND CONSTITUENT CONDITION SPECIFICATIONS
7 FF
ARROW (-->), SEPARATING LEFT AND RIGHT PARTS OF A RULE.
8 09
(708) 1 BA
STYPE OF FIRST CONSTITUENT OF RIGHT
2 6F
3 02
4 8B
5 21
CREATION SPECIFICATIONS FOR FIRST CONSTITUENT ON RIGHT
6 FF
END OF RULE
7 00
8 00
(2269) 1 00
XATWRD (TOP WORD OF STORAGE)
2 00
3 FF
4 0C
5 B9
6 FF
XATBYT (TOP BYTE OF STORAGE)
7 00
8 00
(5000) 1 00
2 00
3 00
4 00
5 00
6 00
7 00
8 00

```
SAMPLE RULE STORED:
     ADV($'FILLER') VERBPH( NOUNAL) -->
                    VERBPH(PRM)
```

Figure 3. Conceptual Structure and Storage of the A-Array

The conceptual structure of the A-array is a LOGICAL*1 (one byte) one-dimensional array of 40,000 bytes. The conceptual structure will be used to describe the storage and retrieval of information from the A-array.

The dimension statements of the A-array and the CELL array can easily be changed within NLP MAIN. Besides changing the dimension statements, MXCELL (maximum subscript of the CELL array) and XMAXA (maximum subscript of the A-array) must also be changed. Thus, only four statements in the MAIN routine need be changed.

2. Left-Part Rule Processor

The following sample decoding rule from NLPQ will be used for illustrative purposes in this section:

```
VERB(MODAL) VERBPH(INF) -->
    VERBPH(PRM,MODAL=MODAL(VERB),VFORM=VFORM(VERB),INTERG)
```

The rule has two segment types on the left of the arrow (NLPQ encoding rules have only one segment type on the left) and one on the right. Each of the segment types on the left has condition specifications in parentheses and the segment type on the right has creation specifications in parentheses. Figure 4A depicts in block diagram format the processing of the left-part of a rule and Figure 4B does the same for the right-part.

PRULES begins compiling rules by first setting some variables, and one of the first variables to be set is XDESW (decoding switch). A XDESW value of "true" indicates that decoding rules are to be processed, and a value of "false"

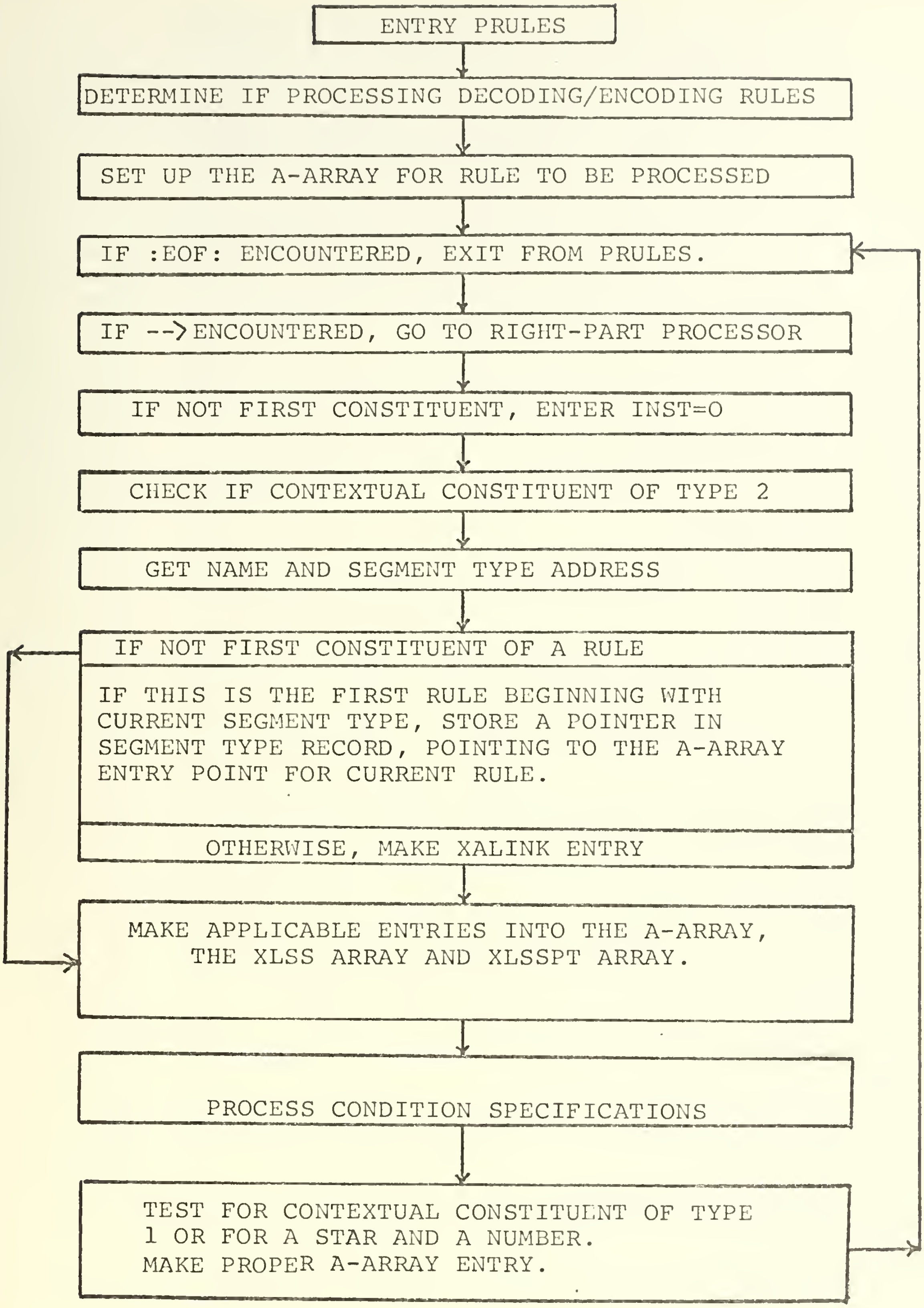

ENTRY PRULES
DETERMINE IF PROCESSING DECODING/ENCODING RULES
SET UP THE A-ARRAY FOR RULE TO BE PROCESSED
IF :EOF: ENCOUNTERED, EXIT FROM PRULES.
IF --> ENCOUNTERED, GO TO RIGHT-PART PROCESSOR
IF NOT FIRST CONSTITUENT, ENTER INST=0
CHECK IF CONTEXTUAL CONSTITUENT OF TYPE 2
GET NAME AND SEGMENT TYPE ADDRESS
IF NOT FIRST CONSTITUENT OF A RULE
IF THIS IS THE FIRST RULE BEGINNING WITH CURRENT SEGMENT TYPE, STORE A POINTER IN SEGMENT TYPE RECORD, POINTING TO THE A-ARRAY ENTRY POINT FOR CURRENT RULE.
OTHERWISE, MAKE XALINK ENTRY
MAKE APPLICABLE ENTRIES INTO THE A-ARRAY, THE XLSS ARRAY AND XLSSPT ARRAY.
PROCESS CONDITION SPECIFICATIONS
TEST FOR CONTEXTUAL CONSTITUENT OF TYPE 1 OR FOR A STAR AND A NUMBER.
MAKE PROPER A-ARRAY ENTRY.

Figure 4A. Prules Entry and Left-Part Rule Processing

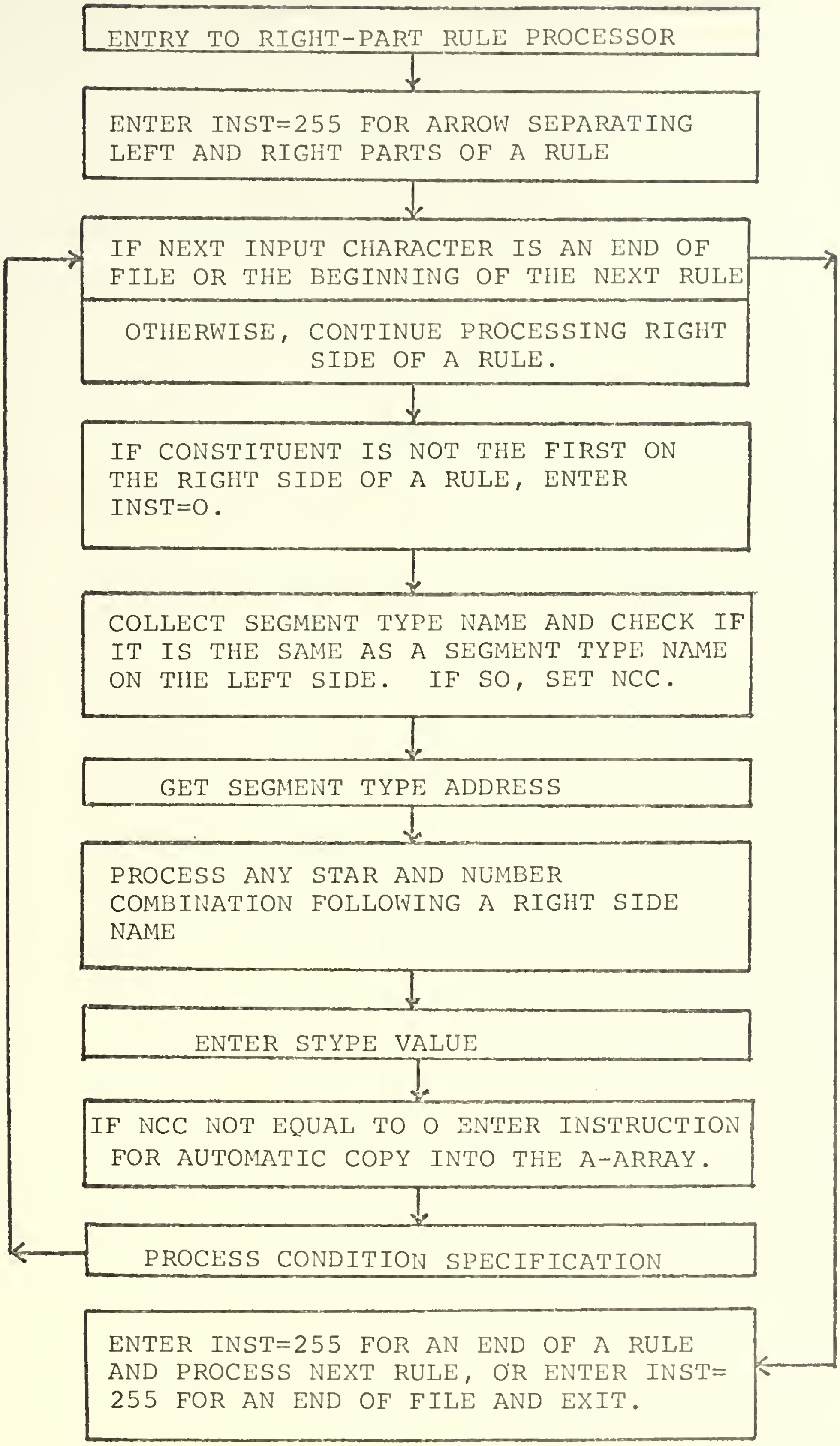

ENTRY TO RIGHT-PART RULE PROCESSOR
ENTER INST=255 FOR ARROW SEPARATING LEFT AND RIGHT PARTS OF A RULE
IF NEXT INPUT CHARACTER IS AN END OF FILE OR THE BEGINNING OF THE NEXT RULE
OTHERWISE, CONTINUE PROCESSING RIGHT SIDE OF A RULE.
IF CONSTITUENT IS NOT THE FIRST ON THE RIGHT SIDE OF A RULE, ENTER INST=O.
COLLECT SEGMENT TYPE NAME AND CHECK IF IT IS THE SAME AS A SEGMENT TYPE NAME ON THE LEFT SIDE. IF SO, SET NCC.
GET SEGMENT TYPE ADDRESS
PROCESS ANY STAR AND NUMBER COMBINATION FOLLOWING A RIGHT SIDE NAME
ENTER STYPE VALUE
IF NCC NOT EQUAL TO O ENTER INSTRUCTION FOR AUTOMATIC COPY INTO THE A-ARRAY.
PROCESS CONDITION SPECIFICATION
ENTER INST=255 FOR AN END OF A RULE AND PROCESS NEXT RULE, OR ENTER INST=255 FOR AN END OF FILE AND EXIT.

Figure 4B. Right-Part Rule Processing

means that encoding rules are to be processed. Next, the first line of the first rule is read. As each new rule is read, it is assigned a rule-number (the value of XATWRD which is a pointer to the top word used in the A-array). For each rule, storage begins in a new word (element) of the A-array.

Once the rule-number has been assigned, the left-part of the rule can be processed. The name of the first constituent is "collected" and stored in NMS (name variable). Then the identifying number (address) of the segment type record for the constituent must be determined. If the name associated with NMS is equal to that associated with NMSS (name variable for name of first constituent from the preceeding rule processed) then STYPE (segment type record address for NMS) is equal to XCSTYP (segment type record address for NMSS). Else, a search must be made by scanning all segment type record names of rules already stored which begin with the first character of the name stored in NMS. If the search indicates that this is the first occurance of such a segment type then a record of this segment type is created in the ISS (Information Storage Structure). STYPE is set to the address of the newly created record. For all segment types which are the first constituent of a rule the content of attribute XATR1 (set to 1 for decoding and 8 for encoding) is assigned to the variable XLRULE. If XLRULE (address of last rule of same segment type) has a value of zero then a pointer is entered which points to a word in the A-array where the first rule

beginning with that segment type will be stored. Also, the variables NMSS, XCSTYP (segment type address for NMSS value) and XLRULE are reset to conform to the constituent currently being processed. If XLRULE contains a pointer to an element of the A-array then the current rule being processed is not the first rule beginning with such a segment type. For this case, the link field (first two bytes of a rule stored in the A-array) of the last rule processed for this segment type must be located and the rule-number of the present rule is inserted. XALINK is the link field variable of a rule. After a link has been made the NMSS, XCSTYP and XLRULE are reset.

All constituents of a rule have their STYPE value entered into the A-array prior to the processing of their condition or creation specifications. This entry is accomplished with a call to CODE using an argument of -4, the one exception being the first constituent of a rule.

Prior to processing conditions the NMS and STYPE values are entered into the XLSS (left side segment type name) array and the XLSSPT (left side segment type pointer) array respectively.

The rule-number of the sample rule is 02B6 (694 decimal) and the first A-array word of the previous rule with the same segment type is:

```
02 B6 DB 05 66 00 09 BA
```

with the link field entry (first two bytes) pointing to the rule currently being processed.

At this stage of processing, a check for a left parenthesis is made. A positive result indicates that the constituent has conditions which must be compiled. PRCNLB, using GETSYM and CODE, acts as a compiler and is called to process these condition specifications. After compiling the first constituent of the sample rule, the content of the A-array for the constituent is:

```
00 00 CD 18 16 00 00 00
```

While the left-part of a rule is being processed, two checks are performed on decoding rules to determine if the constituent being processed is a contextual constituent. There are two types of contextual constituents. A type 1 is indicated by a slash (/) appearing after a constituent in the rule. A slash appearing next to the beginning of a constituent signifies a type 2 contextual constituent. The checks are performed as the rule is scanned from left-to-right, and any contextual information is entered into the A-array with a call to CODE using an argument of -3. The variable NUMB contains the contextual type value.

A constituent can also be followed by a star (*) and a number. If the number is missing then a 3 is used as the default value. Such information is entered into the A-array with the same call to CODE as for contextual information. A constituent can not have both a slash and a star associated with it. The variable NUMB is set to the number following the star. The call to CODE is made after the condition specifications, if any, have been processed.

When a constituent has been processed, program execution returns to the beginning of the left-part processor. The next group of characters in the rule are located and if the group is an arrow then control is switched to the right-part processor. Otherwise, the next constituent of the rule has been located, as is the case for the example, and a zero-value byte is entered into the A-array with a call to CODE using an argument of -2. The zero-value byte has the purpose of separating constituents. The second and any following constituents are processed much like the first except that NMSS, XCSTYP, and XLRULE are not reset, and linking is not performed.

When the arrow is encountered, the left-part of a rule has been processed and control of program execution is transferred to the right-part processor. The first action of the right-part processor is to enter the value 255 into the A-array to indicate the separation between the two parts. This is accomplished with a call to CODE using an argument of -5.

After processing the left-part of a rule, the A-array content for the sample rule is:

```
00 00 CD 18 16 00 09 BA
8D 10 FF 00 00 00 00 00
```

3. Right-Part Rule Processor

The right-part processor initially performs the same functions as the left-part processor. That is, the NMS (segment type name) of the constituent must be collected and

its STYPE (segment type address) determined. Since the NMS value of a constituent on the right can equal one already processed on the left, the XLSS array is searched first. If the search is successful, the STYPE value is obtained from the corresponding XLSSPT array entry. Otherwise, a search of segment type records, the same as performed by the left-part processor, is made. If no match is found, a record for the segment type is created in the Information Storage Structure and STYPE is set to point to the new record. If the segment type name is followed by a star and a number, the value of AMCL (attribute four) of the record pointed to by STYPE, is set to the number.

All constituents on the right have their STYPE value entered into the A-array. Also, for any NMS value equal to an entry in the XLSS array a special entry (instruction code 47) is made in the A-array. This special entry specifies that the particular constituent being processed is to be a copy of some constituent on the left. For decoding rules a number is inserted into the A-array which specifies which left constituent is to be copied during execution of the rule. This information is entered with a call to CODE using an argument of -1. For encoding rules there is only one constituent on the left and thus is is not necessary to enter a number.

For encoding rules the last constituent on the right whose NMS value is equal to the NMSS (same as XLSS(1)) value, does not require that a copy of the segment record from the left be made during execution of the rule. Instead, the

right side constituent can use the same segment record as the one on the left, making changes as designated by the creation specifications. To indicate that such a situation exists, the instruction code of 47 which was set for the right side constituent is changed to an instruction code of 52 in this case.

If the constituent currently being processed has any creation specifications, then a call to PRCNLB will compile those specifications. After returning from PRCNLB, control of processing is transferred to the beginning of the right-part processor which will test if there are more constituents on the right or if another rule follows. If there are more constituents, each is separated in the A-array by a zero-value byte. If there is another rule, then control of the program is transferred to the PRULES portion which processes the next rule. If there are no more rules then a normal return from PRULES is made.

When returning from PRULES, the sample rule will have been compiled and stored in the A-array in the following manner:

```
00 00 CD 18 16 00 09 BA
8D 10 FF 09 BA 6F 02 8B
21 5E 01 D4 18 16 C9 18
16 5E 01 D4 13 0D C9 13
0D 6B 1F FF 00 00 00 00
```

C. PROCESSING CONDITION AND CREATION SPECIFICATIONS (PRCNLB)

Whenever a rule or named record definition has information within parentheses, PRCNLB is called to process that information. (i.e. translate it into a series of elementary commands which can later be executed by TSTCND or CRSEG.) The function of PRCNLB within the compiling process is as a stacking decision procedure. While performing this function PRCNLB repeatedly calls GETSYM which returns a TSCODE and TSADDR value for each input symbol recognized by GETSYM. PRCNLB, by using TSCODE, decides whether to stack TSCODE and TSADDR onto the SCODE and SADDR vectors, respectively, or to cause a reduction of the vectors with a call to CODE which processes the information in SCODE and SADDR, storing the results in the A-array.

A normal return from PRCNLB occurs when a right parenthesis is recognized as an input symbol.

D. GETTING NEXT INPUT SYMBOL (GETSYM)

Input symbols recognized by NLP are all the letters, digits, and most of the special symbols found on an IBM 29 card punch. In addition the system will process as symbols the six standard arithmetic logicals when between two periods (e.g. .LT.), names of up to eight characters, and numbers. Also, names with eight or fewer characters within single quotation marks and EBCDIC strings of any length within double quotation marks. The special symbols which can be recognized by GETSYM are found in the SCDTAB array and indicated by an entry value greater than zero.

The function that GETSYM performs within the compiling procedure is that of a lexical analyzer. GETSYM collects the input characters and translates them to symbol codes (TSCODE) with associated values (TSADDR). Appendix E contains a list of symbol codes.

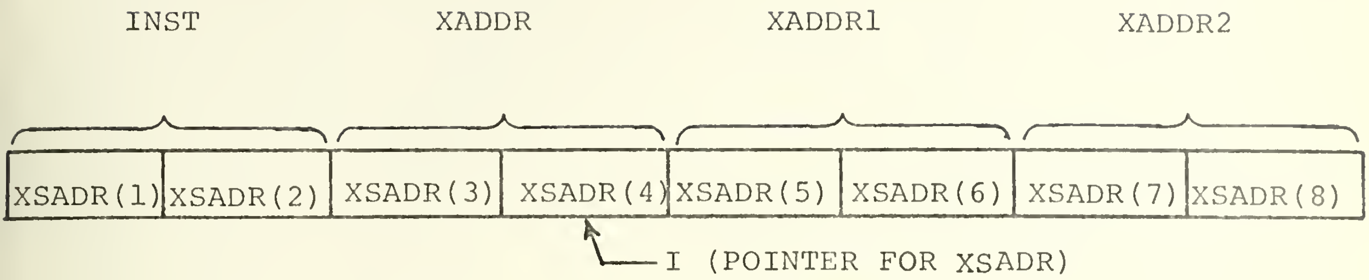
E. PRODUCING CODE FOR CONDITION AND CREATION SPECIFICATIONS (CODE)

The reduction function within the compiling procedure is performed by CODE. When CODE is called from PRCNLB, reduction on the content of SCODE and SADDR occurs. When the call is from PRULES, only specified information is entered into the A-array.

All the calls from PRULES use negative arguments. A -1 enters an instruction code of 47 into the A-array followed by the NCC (number of constituent to be copied) value as set by PRULES. A -2 enters into the A-array a zero-value byte which separates the constituents of a rule. A -3 enters an instruction code of 51 into the A-array. This is the instruction code for setting the contextual constituent type. The entry is followed by the NUMB value as set by PRULES. For an argument of -4 an instruction code of 194 is created which has the purpose of setting aside two bytes in the A-array into which will be entered a segment type address (all segment type addresses use two bytes of storage). The -5 argument has the effect of entering the value 255 (FF hexadecimal) into one byte of the A-array.

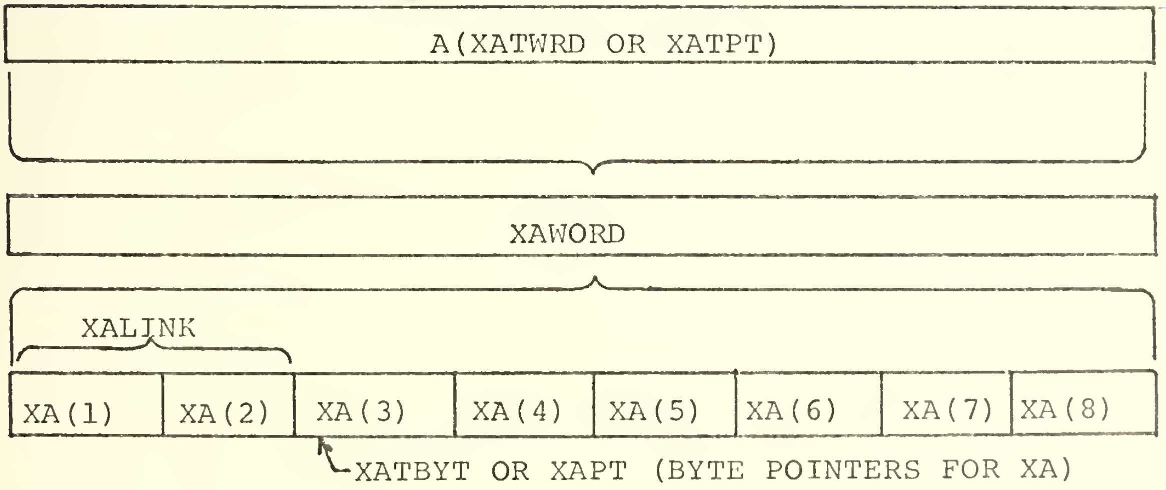
The A-array is initialized by NLP MAIN from a specified input file, or internally with all array elements set to zero. Information which is to be stored by CODE is contained in the variables INST and XADDR. These are INTEGER*2 variables which must be separated into single byte components because information is stored into the A-array in byte format. Figure 5A shows how EQUIVALENCING is used to separate INST and XADDR into their respective XSADR byte components. XSADR is a LOGICAL*1 array of eight elements. XADDR1 and XADDR2 are INTEGER*2 variables whose values are usually equal to the first and second bytes of XADDR. When CODE has determined which bytes of XSADR are to be stored into the A-array, then the contents of those bytes is transferred into XA array. Figure 5B shows the EQUIVALENCING of XA and XAWORD, and an A-array element A(XATWRD) as it relates to XAWORD. XA is a LOGICAL*1 array of eight elements. It receives the byte values to be stored from XSADR, and through XAWORD (one eight-byte word) enters one word of information into A(XATWRD) (the A-array element pointed to by the top word used pointer). XATBYT is a pointer which keeps track of which byte in XA is the top byte used.

As will be seen in the next chapter, to retrieve information from the A-array the above procedure is reversed. But, before information can be retrieved, pointers to the desired A-array word and byte must be set. The pointers for retrieval of information are XATPT and XAPT. XATPT points to the A-array word and XAPT points to the desired byte in the XA array.



```
      TO SET XADDR1:         XSADR(6)=XSADR(3)

      TO SET XADDR2:         XSADR(8)=XSADR(4)

    EQUIVALENCE   (INST,XSADR(1)),(XADDR,XSADR(3)),
                     (XADDR1,XSADR(5)),(XADDR2,XSADR(7))
```

Figure 5A. Equivalencing XSADR,INST, XADDR,XADDR1,XADDR2



```
EQUIVALENCE   (XAWORD,XALINK,XA)
```

Figure 5B. Equivalencing XAWORD,XALINK,XA

When processing information in SCODE and SADDR, CODE begins with the last entry in SCODE and eventually ends with the first entry. SCODE values, which are symbol codes, specify to CODE what meaning input symbols to NLP have. As each symbol code is removed from SCODE, a branch is made to the fortran statements which process the code. The fortran statements within CODE translate the symbol codes into appropriate instruction codes. A list of instruction codes and their meaning is shown in Appendix F. Once INST (variable initially set to an instruction code) and XADDR (variable whose content is set equal to the SADDR value currently being processed) are properly set, CODE enters their respective values into the A-array. Figure 6 shows a flow chart of how information is entered into the A-array.

Before CODE can store information into the A-array a determination must be made as to how many bytes of storage are required for INST and XADDR. First, the space requirements for XADDR must be determined because the actual instruction code which is finally stored depends on the XADDR value. The range of NLP instruction codes is from 1 to 63. However, depending upon the number of bytes required for XADDR, multiples of 64 are added, extending the range of INST from 1 to 255. An INST value less than 64 signifies that no bytes are required for XADDR. An INST value between 64 and 127 means that one byte of storage is required. A value between 128 and 191 means that the two bytes of XADDR are equivalent and thus only one byte need be stored. For values greater than 191 the two bytes in the A-array following

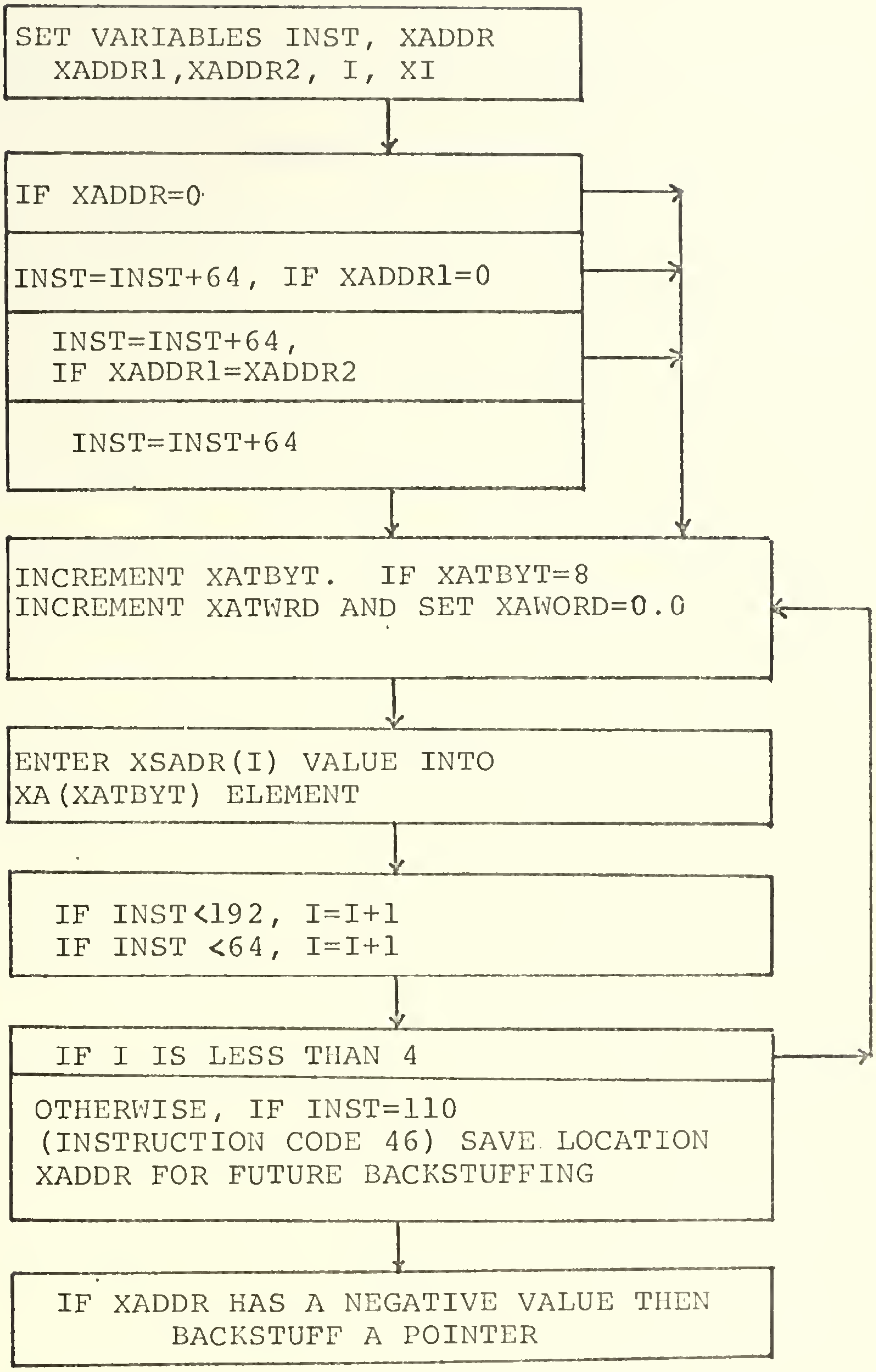

SET VARIABLES INST, XADDR
XADDR1,XADDR2, I, XI
IF XADDR=0
INST=INST+64, IF XADDR1=0
INST=INST+64,
IF XADDR1=XADDR2
INST=INST+64
INCREMENT XATBYT. IF XATBYT=8
INCREMENT XATWRD AND SET XAWORD=0.0
ENTER XSADR(I) VALUE INTO
XA(XATBYT) ELEMENT
IF INST<192, I=I+1
IF INST <64, I=I+1
IF I IS LESS THAN 4
OTHERWISE, IF INST=110
(INSTRUCTION CODE 46) SAVE LOCATION
XADDR FOR FUTURE BACKSTUFFING
IF XADDR HAS A NEGATIVE VALUE THEN
BACKSTUFF A POINTER

Figure 6. Storage Procedure for the A-Array

the INST value contain the XADDR value. The one exception is an INST value of 255 which does not have any associated XADDR value.

An INST value of 46 also requires some special processing. XADDR for such an INST will require one byte of storage, but the XADDR value to be stored is not available at the time INST 46 is put out. Thus, INST value 46 is coded to 110 and a zero-value byte is entered for the XADDR value. The A-array location of this zero-value byte is set by the value of XCURNT (contains the current absolute byte address in the A-array) and is saved in the ATSTOR (attribute store) array. The SCODE value is changed to 7 and the respective SADDR location is set to the negative value of ATI (current ATSTOR subscript). When the proper XADDR value becomes available, the zero-value byte address stored in ATSTOR is retrieved and a pointer value entered at the byte address location. The value which is backstuffed is a pointer to an A-array location where the proper XADDR value associated with INST value 46 can be found. The pointer value is the relative number of bytes from the zero-value byte to where the XADDR value is stored. An example of when INST value 46 occurs is:

```
(@N=5)
```

The "@" symbol signifies that the attribute whose number is in N be set equal to 5.

F. PROCESSING OF NAMED RECORD DEFINITIONS (PRNREC)

Named record definitions look very similar to the constituents of a rule and are processed in much the same manner, except that PRNREC is used instead of PRULES. The main difference is that the code generated from the creation specifications is executed immediately by CRSEG, rater than simply left in the A-array to be used later. Figure 7 shows in flow chart form how named record definitions are processed. A detailed discussion of named record definitions can be found in Ref. 4.

PRNREC collects the name and creates a record in the CELL array. XATPT and XAPT are respectively set to the top available word and byte in the A-array. When a left-parenthesis is encountered the information in parentheses is processed by PRCNLB.

To enter the attribute specifications into CELL array, PRNREC calls CRSEG which executes the code just compiled for the named record definition and stored in the A-array, as will be discussed in the next chapter. After returning from CRSEG the next named record definition can be processed, and the content of the A-array for the previous named record definition is no longer needed, and can be erased. This is accomplished by resetting XATWRD to the value which it contained when entering PRNREC, and resetting XATBYT to zero. This allows the next named record definition to use the same A-array storage locations that the previous named record definition used.

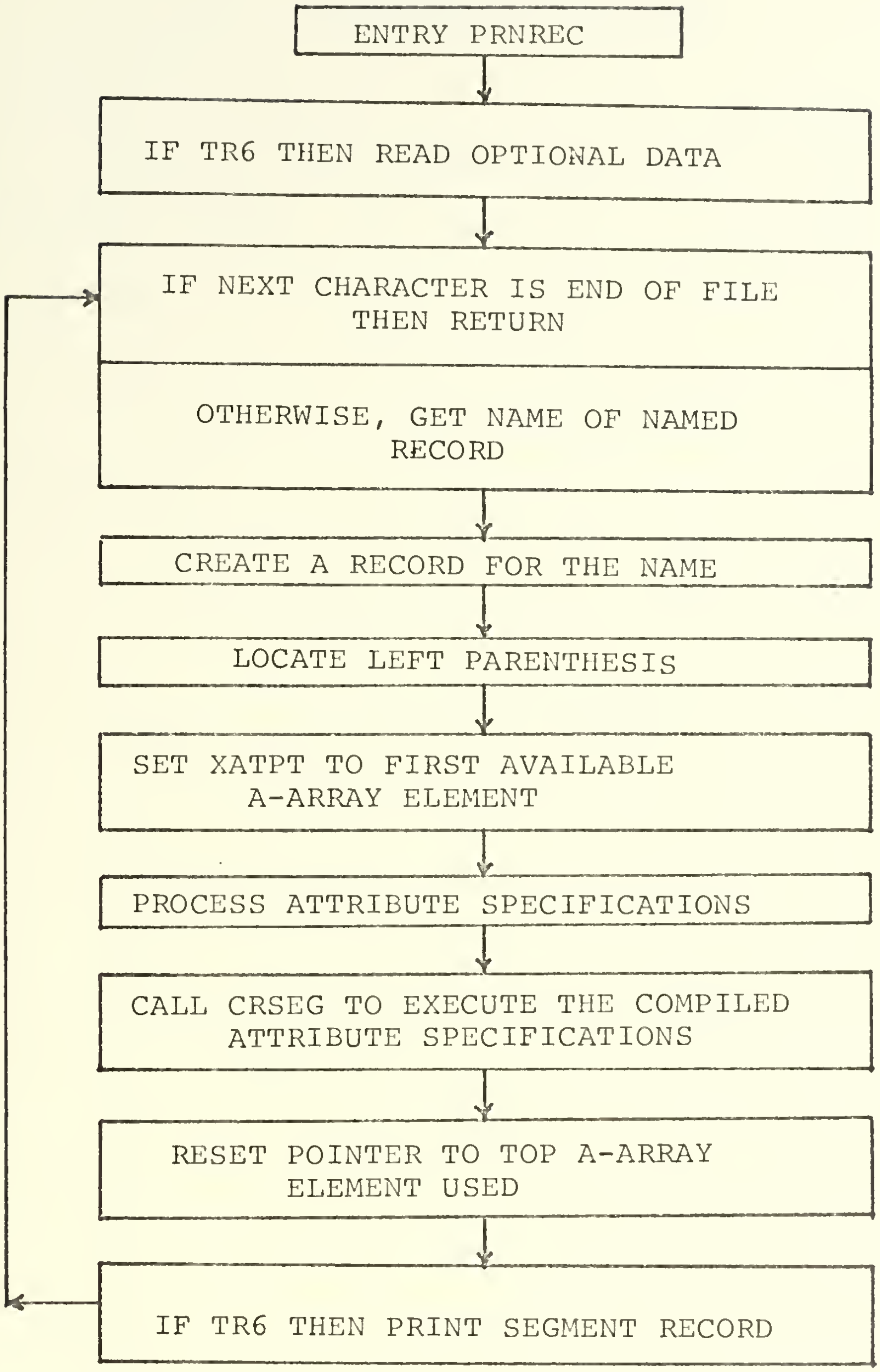

ENTRY PRNREC
IF TR6 THEN READ OPTIONAL DATA
IF NEXT CHARACTER IS END OF FILE THEN RETURN
OTHERWISE, GET NAME OF NAMED RECORD
CREATE A RECORD FOR THE NAME
LOCATE LEFT PARENTHESIS
SET XATPT TO FIRST AVAILABLE A-ARRAY ELEMENT
PROCESS ATTRIBUTE SPECIFICATIONS
CALL CRSEG TO EXECUTE THE COMPILED ATTRIBUTE SPECIFICATIONS
RESET POINTER TO TOP A-ARRAY ELEMENT USED
IF TR6 THEN PRINT SEGMENT RECORD

Figure 7. Processing of Named Record Definitions

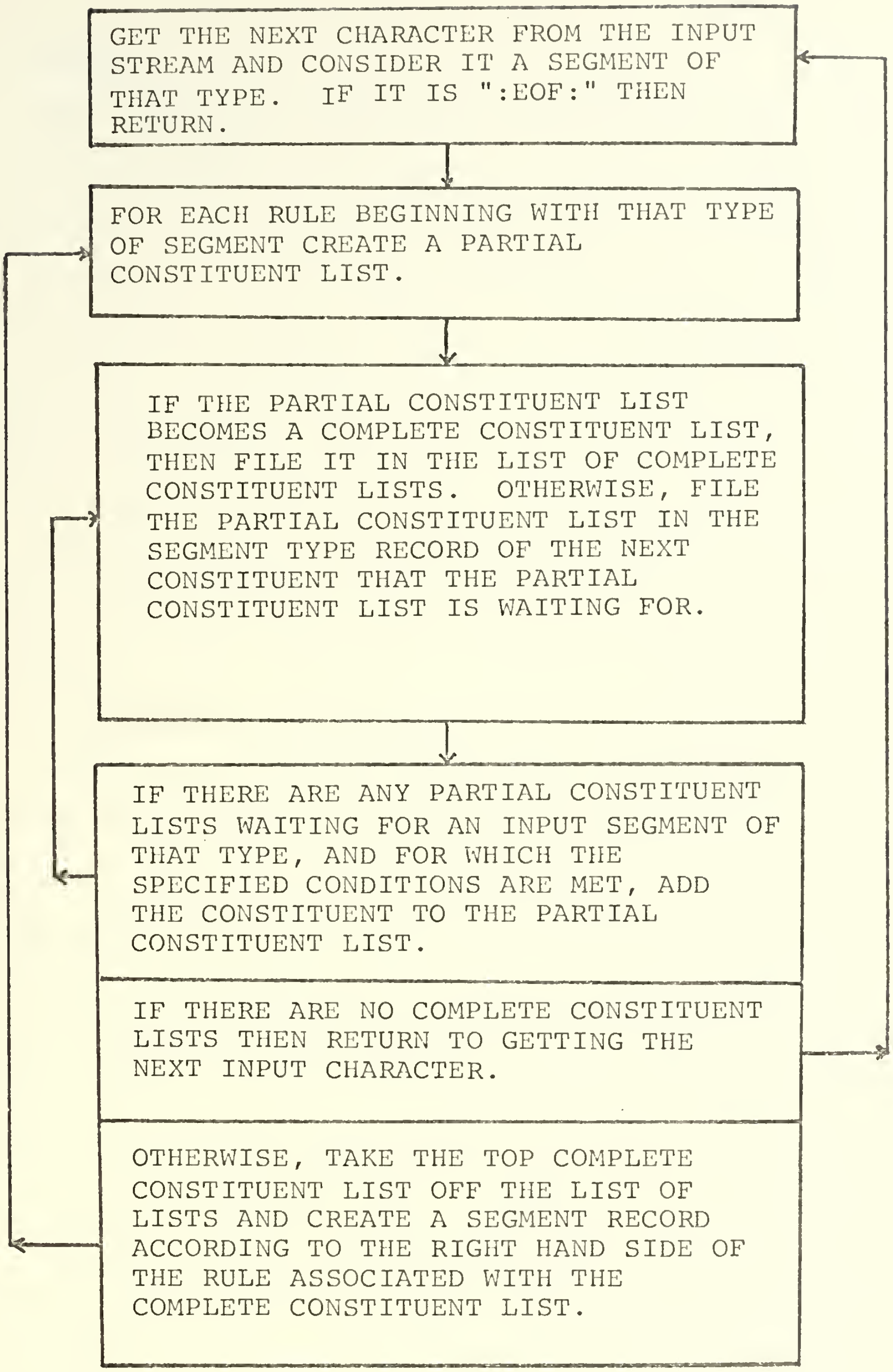

GET THE NEXT CHARACTER FROM THE INPUT STREAM AND CONSIDER IT A SEGMENT OF THAT TYPE. IF IT IS ":EOF:" THEN RETURN.
FOR EACH RULE BEGINNING WITH THAT TYPE OF SEGMENT CREATE A PARTIAL CONSTITUENT LIST.
IF THE PARTIAL CONSTITUENT LIST BECOMES A COMPLETE CONSTITUENT LIST, THEN FILE IT IN THE LIST OF COMPLETE CONSTITUENT LISTS. OTHERWISE, FILE THE PARTIAL CONSTITUENT LIST IN THE SEGMENT TYPE RECORD OF THE NEXT CONSTITUENT THAT THE PARTIAL CONSTITUENT LIST IS WAITING FOR.
IF THERE ARE ANY PARTIAL CONSTITUENT LISTS WAITING FOR AN INPUT SEGMENT OF THAT TYPE, AND FOR WHICH THE SPECIFIED CONDITIONS ARE MET, ADD THE CONSTITUENT TO THE PARTIAL CONSTITUENT LIST.
IF THERE ARE NO COMPLETE CONSTITUENT LISTS THEN RETURN TO GETTING THE NEXT INPUT CHARACTER.
OTHERWISE, TAKE THE TOP COMPLETE CONSTITUENT LIST OFF THE LIST OF LISTS AND CREATE A SEGMENT RECORD ACCORDING TO THE RIGHT HAND SIDE OF THE RULE ASSOCIATED WITH THE COMPLETE CONSTITUENT LIST.

Figure 8. Decoding Algorithm

1. DECODE

When NLP main recoginzes the command DECODE: (or TEXT:), routine DECODE is called to process the input text which follows. But before the input text is read, DECODE enters a period (.) and a blank (signified by #) at the front of the input stream. This has the function of setting up the proper variables which will expect the input text to be the start of a new sentence.

Before any text processing is performed, DECODE will request optional data if switch TR6 was set to true, allowing the user to set the program parameters and switches as listed in Appendices B and C.

To begin processing, DECODE removes from the input stream the first character. At this time the input stream contains a period and a blank. Decode processes the two characters and thereby sets the stage for processing input text. After the first line of text is read the input stream may have the following characters for example:

```
.#VEHICLES#ARRIVE#AT#A#STATION.
```

DECODE obtains the next character from the input stream and considers it to be a segment of that type. Then NEWSEG and ADDSEG are called to process the character, providing DECODE with a list of complete constituent lists. These complete constituent lists contain all the instances of the decoding rules which have had their left-part conditions

satisfied with the appearance in the input stream of this particular character. If the list is empty, then the next input character is processed. Otherwise, DECODE removes the complete constituent lists from the list of lists, one at a time, and creates segment records as specified by the right-part of the rule whose conditions were met.

The rules which are to be executed are stored in the A-array as they were compiled, and must be retrieved by a process which is essentially the reverse of that discussed in section III.E. The complete constituent list contains the XATPT and XAPT values which point to the byte in the A-array where the information concerning the right-part of a rule begins. The first two bytes retrieved contain the segment type record address (STYPE value) of the first constituent on the right. The execution of the creation specifications (information in parentheses for segment types on the right-part of a rule) is performed by CRSEG which is called from NEWSEG, which in turn is called from DECODE. For each constituent on the right side of the rule, DECODE sets the STYPE variable and calls NEWSEG. If while processing the right-part of a rule the value 255 is encountered, the next entry on the list of complete constituent lists is processed. When all such entries have been processed, the next character from the input stream is obtained and processed. When an end of file symbol (:EOF:) is encountered in the input stream, DECODE returns control to NLP MAIN.

2. NEWSEG

When NEWSEG is called, it has the function of investigating the new segment type under consideration. This investigation consists of locating all the rules which begin with a segment of this type. A pointer to the first such rule is obtained from the APFCR attribute of the segment type record, and each rule points to the next one, as will be described below. Each rule on the list has its first constituent condition specifications checked by TSTCND. If the condition specifications are met, NEWSEG creates a partial constituent list, which serves as an indicator of what state of recognition a particular instance of a rule is in. ADDSEG will determine if the status of the rule is to "wait" for another constituent or to "yield" a new segment type (on the right).

After returning from the call to ADDSEG the next rule beginning with a segment of the current type is processed. NXTRL contains the address of the next rule to be processed. NXTRL obtains its value from the XALINK (link to the next rule of same segment type) field of the previous rule. When NXTRL equals zero, the end of the list has been reached.

Once the list of rules beginning with a segment of the current type is exhausted, the partial constituent lists waiting for a segment of that type are processed. First, the list of those waiting for a segment of the current type are located. Then, only those for whom the segment occurs in the proper position in the input are processed. If the condition

specifications for this constituent are met, the segment is added to the partial constituent list by a call to ADDSEG.

3. ADDSEG

With each segment to be added to a partial constituent list ADDSEG is called. ADDSEG determines if the partial constituent list should be filed in the list of complete constituent lists (if there are no more constituents in the rule), or if a pointer should be filed in the segment type record of the next constituent of the rule. The second process has the effect of signaling for which input segment a rule is waiting. If the input segment satisfies the last condition of a rule then the partial constituent list becomes a complete constituent list and yields a new segment type.

B. ENCODING PROCESS

The encoding process provides the user with output for some given input. If the input was a natural language description of a queuing problem, as is the case for NLPQ, the output could be a GPSS program. The execution of the encoding rules specify what the output will be.

Figure 9 is a flow chart of the encoding algorithm. The algorithm is executed when the encoding command is recognized. The form of the encoding command can be a sentence describing what encoding action is to be performed. Two examples are:

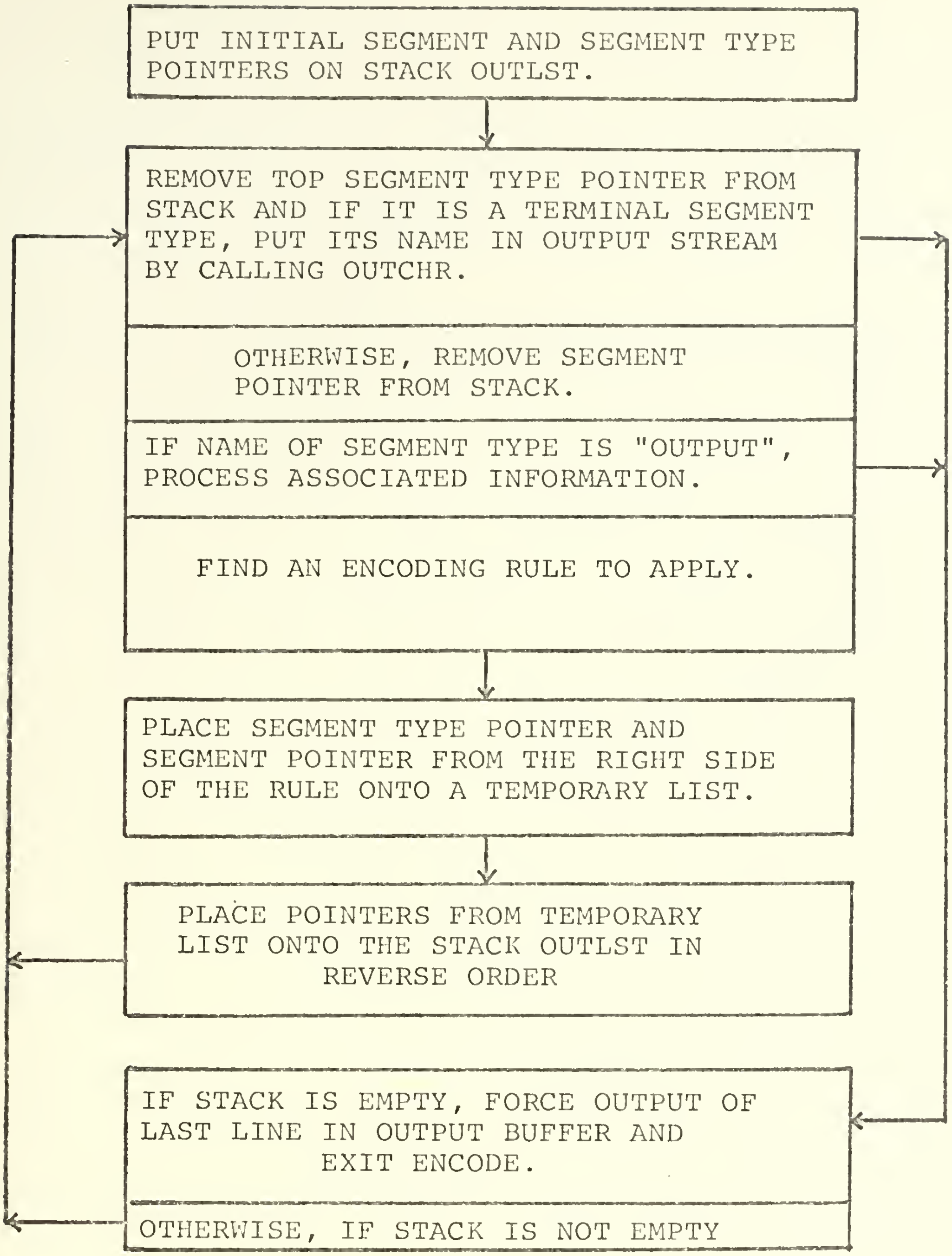

PUT INITIAL SEGMENT AND SEGMENT TYPE POINTERS ON STACK OUTLST.
REMOVE TOP SEGMENT TYPE POINTER FROM STACK AND IF IT IS A TERMINAL SEGMENT TYPE, PUT ITS NAME IN OUTPUT STREAM BY CALLING OUTCHR.
OTHERWISE, REMOVE SEGMENT POINTER FROM STACK.
IF NAME OF SEGMENT TYPE IS "OUTPUT", PROCESS ASSOCIATED INFORMATION.
FIND AN ENCODING RULE TO APPLY.
PLACE SEGMENT TYPE POINTER AND SEGMENT POINTER FROM THE RIGHT SIDE OF THE RULE ONTO A TEMPORARY LIST.
PLACE POINTERS FROM TEMPORARY LIST ONTO THE STACK OUTLST IN REVERSE ORDER
IF STACK IS EMPTY, FORCE OUTPUT OF LAST LINE IN OUTPUT BUFFER AND EXIT ENCODE.
OTHERWISE, IF STACK IS NOT EMPTY

Figure 9. Encoding Algorithm

Describe the problem in English.

Write a GPSS program for this problem.

The "sentence" commands are processed by the decoding algorithm, which then calls ENCODE after recognizing what action is to be performed.

ENCODE is called with two arguments, STYPEE and SGMNTE. STYPEE is a segment type pointer and SGMNTE is a segment pointer. First, SGMNTE is entered on the OUTLST stack and then STYPEE is entered. Next, a segment type pointer is removed from the top of the stack. If the segment is a terminal segment type, its name is placed in the output stream with a call to OUTCHR. Otherwise, a segment pointer is removed from the top of the stack. If the segment type name associated with the segment pointer is "output" then more information is usually entered into the output stream. If the name is not "output", then testing the left-part conditions of encoding rules beginning with such a segment type name begins. If a rule has its conditions satisfied then the segment type pointer and a segment pointer of the first constituent on the right are placed on a temporary list. Then the pointers from the temporary list are placed onto OUTLST, in reverse order.

When all the entries on the OUTLST stack have been processed, the output of the last line in the buffer is forced out and a return from ENCODE is made.

C. TESTING CONDITIONS AND CREATING RECORDS (TSTCND AND CRSEG)

Routines TSTCND and CRSEG share the same fortran code. The difference between the two routines is that CRSEG initializes the CNLB (condition or label) variable to 0 and obtains an address for SEGMNT if there is none. TSTCND sets CNLB to 1. In function, the two routines are very different. CRSEG executes compiled rules creating segments as specified by the creation specifications of the right-part of a rule. TSTCND tests the condition specifications on the left-part of a rule.

CRSEG is called from NEWSEG, ENCODE and PRNREC, while TSTCND is called from NEWSEG and ENCODE. The instruction codes processed by CRSEG and TSTCND are listed in Appendix F.

For the remainder of this section, a reference to CRSEG will also imply TSTCND unless specifically stated as being otherwise.

When CRSEG is called, all the instruction codes to be executed are contained in the A-array. Thus the first function of CRSEG is to retrieve the INST and XADDR values from the A-array. Once ATC (equal to INST+1) and ADC (equal to XADDR) are set, the specific instruction codes can be executed by the remainder of CRSEG.

Before CRSEG is called, XATPT and XAPT must be set to point to the proper byte in the A-array where interpretation is to begin. INST is set equal to the information contained in the first byte to be processed. Each time INST is set, it is tested for a zero value (separator between constituents) or a value of 255 (represents the "arrow" or the end of a

rule). If either of these values is encountered, processing of instruction codes ceases, and a return is made (the one exception being if XI (transfer variable) is set to 3). Otherwise, each INST and associated XADDR is processed.

CRSEG reverses the storage procedure for INST and XADDR as performed by CODE. In CRSEG the INST value obtained from the A-array specifies how many bytes following INST must be processed to get the proper XADDR value. EQUIVALENCING as shown in Figures 5A and 5B, and as already explained, is used to process the A-array content. Once the proper INST and XADDR values are obtained, ATC and ADC are set. The ATC value determines where within CRSEG control is transferred to process the particular INST code. After each INST code and XADDR have been processed, the next INST is obtained from the A-array. This is performed by setting I (subscript variable for XSADR) equal to 2, XI equal to 1 and extracting the next byte from the A-array. This process continues until an INST value of 0 or 255 is encountered. For TSTCND, either of these instruction codes (0 or 255) will cause a return from the routine. For CRSEG, if the calling argument was not 0, some final entries for the newly created segment record are made and then a return is made.

There are instances during the execution of CRSEG when a return is signaled before an INST value of 0 or 255 is encountered. Also, during decoding the situation can arise when it is desirable to "skip" to the end of parentheses for a constituent being processed. If either of these conditions is encountered the EXECSW (execute switch) is set to false.

Setting EXECSW to false causes CRSEG to continue retrieving INST and XADDR values until A 0 or 255 INST value is encountered. The instruction codes retrieved with EXECSW set to false will not be executed except for INST value 51. The result of such processing is to exit CRSEG with XATPT and XAPT pointing to the proper A-array byte for the decoding or encoding process to continue.

V. RESULTS

To be able to determine the effectiveness of implementing the A-array, the sample NLPQ terminal sessions described in Ref. 3 were repeated. Also, the decoding, massager, English encoding and GPSS encoding rules were compiled, and the named record definitions were processed for the comparisons.

Implementing the A-array had the primary objective of reducing the storage requirements for the rules used with NLP. Table 1 contains the results of comparing the old and new methods of storing NLPQ rules. SET1, SET2, SET3, and SET4 in Table 1 respectively represent the four sets of rules mentioned above. The leftmost column of Table 1 has seven entries representing the old and new storage requirements as related to NLPQ rules.

The first three rows of Table 1 contain the number of storage elements required by the old storage method. The next three rows represent the rule storage requirements with the new method. Row 7 shows the number of 8-byte elements saved with the new method of storing NLPQ rules. The first

		RULES				
		SET1	SET2	SET3	SET4	TOTAL
	STORAGE SPACE					
OLD	RECORDS(CELL ARRAY)	802	162	368	491	1823
	RULES(CELL ARRAY)	6515	772	3049	1843	12179
	TOTAL ELEMENTS	7317	934	3417	2334	14002
NEW	RECORDS(CELL ARRAY)	802	162	368	491	1823
	RULES(A-ARRAY)	1354	181	735	457	2727
	TOTAL ELEMENTS	2156	343	1103	948	4550
	SAVINGS	5161	591	2314	1386	9452

Table 1. Old and New NLPQ Rule Storage Requirements (Numbers Represent 8-Byte Elements)

and fourth rows contain the number of CELL array record elements that result from compiling the rules. These elements are primarily for segment type records and named records not previously defined. Row 2 shows the number of CELL array storage elements the NLPQ rules required under the old storage method. The addition of values from rows 1 and 2 gives the total CELL array requirements for the old method, and these values are shown in Row 3. The fifth row shows the new storage requirements when the rules are stored in the A-array. The sixth row shows the total number of 8-byte elements required using the new scheme. The savings of 8-byte elements accomplished for rule storage are shown in row 7. These values are the differences between rows 2 and 5. The percentage of 8-byte element savings for the rules alone is 78% and the percentage for the total number of elements saved is 67%.

The secondary objective of this thesis was to reduce the amount of paging performed by CP/CMS while executing NLP. Such a reduction is reflected in the actual CPU times shown in Table 2. A reduction in the virtual CPU time was also achieved, as can be seen in the Table, probably due to less list processing. Columns 1 and 2 show the virtual CPU times for the old and the new methods and columns 3 and 4 show the actual CPU times.

The upper part of the table lists the CPU times taken for compiling of NLPQ rules and processing the named record definitions. The total times for these two are shown in row 7. The values show that the new method saved 31 seconds of

	TIMES (IN SECONDS)			
	VIRTUAL		ACTUAL	
	CPU TIME		CPU TIME	
	OLD	NEW	OLD	NEW
RULES:				
SET1	46	37	108	64
SET2	13	9	28	18
SET3	32	24	86	54
SET4	25	16	65	38
TOTAL FOR RULES	116	86	287	174
NAMED RECORD DEFINITIONS	16	15	38	30
TOTAL COMPILING	132	101	325	204
SAMPLE PROBLEM:				
DECODING	142	133	315	218
ENGLISH ENCODING	25	26	76	41
GPSS ENCODING	24	22	64	48
TOTAL EXECUTING	191	181	455	307

Table 2. Virtual and Actual CPU Times for the Old and New Method for Processing the Sample NLPQ Problem.

virtual CPU time and 121 seconds of actual CPU time. The respective savings percentages accomplished are 24% and 37%.

The lower part of Table 2 shows the times required for executing the NLP program with the sample problem as input. Row 8 shows the times for decoding the problem into the IPD. Row 9 shows the times for encoding the Internal Problem Description (IPD) into an equivalent English problem description. Row 10 shows the times for encoding the IPD into a GPSS program. Row 11 shows the total execution times for both the old and new methods. The virtual CPU time savings is 10 seconds and the actual CPU time savings is 138 seconds. The respective percentages of time saved are 5% and 31%.

VI. CONCLUSIONS AND RECOMMENDATIONS

The eventual implementation of a general purpose natural language processor is inevitable. Such a processor may not be general in the sense that any input statement will be processed properly, but at least it will be very flexible within its specific application. NLP is a significant advance in this area of natural language processing, and NLPQ is a working example of natural language man-machine interaction using NLP.

As such processors become more flexible, it can be expected that they will require more decoding and encoding rules. The storage scheme developed and discussed in this thesis has significantly reduced the amount of core storage needed to store these rules, as discussed in Section V. As more rules are added, the flexibility of specific NLP applications will increase. The associated time savings, also discussed in Section V, make NLP more responsive to the user, in addition to reducing the computer cost when operating the system.

For further reductions of storage requirements it is recommended that techniques developed in this research be applied to the storage of other information in NLP, such as partial constituent list records and segment type records. Also, if NLP continues to be used with a time sharing system, work should be done to reduce the amount of paging performed. Such a reduction would make NLP even more responsive to the user.

APPENDIX A

NLP COMMANDS

COMMAND	DEFINITION
ATTRIBUTES:	process attribute names
CONVERSE:	perform a special kind of encoding
DECODE:	perform decoding of text
DELETE:	delete specified rules
ENCODE:	perform encoding
END:	terminate NLP program
INDICATORS:	process indicator names
LEXOLOGY:	process lexological decoding rules
LEXOLOGY FOR ENCODING:	process lexological encoding rules
MAXLN:	print number of CELL array elements used for NLP processing
MORPHOLOGY:	process morphological decoding rules
MORPHOLOGY FOR ENCODING:	process morphological encoding rules
NAMED RECORDS:	process named record definitions
OPDATA:	read optional data
PRINT:	print specified information
ROUTINES:	process routine names
SEMOLOGY:	process semological decoding rules
SEMOLOGY FOR ENCODING:	process semological encoding rules
SETRLEV:	set trace level of specified segment type
SIMULATE:	call the simulation subroutine
TEXT:	perform decoding of text
UCELLS:	print number of cell array elements used for storage

APPENDIX B

PARAMETERS

(Default Values are given in parentheses)

ENCNMS	(0.0) segment type to be encoded during decoding
OUTFLA	(6) output file for printed encoded text
OUTFLB	(0) output file for encoded text to be punched
OUT6	(6) output file used for most output
PRCNMS	(0.0) segment type to print constituent structure
PRSLVL	(1) depth of record printout, used with PRSNMS
PRSLVT	(1) depth of record printout, used with PRSNMT
PRSNMS	(0.0) segment type to be printed during decoding or encoding
PRSNMT	(0.0) segment type to be printed during decoding or encoding
RTERM	(5) file number for terminal input
TRINDX	(0 or 1 if TR6 is on) index at which optional data is to be entered
TRLEVS	(100) trace level of segment types to be traced during decoding
WTERM	(6) file number for terminal output

APPENDIX C

SWITCHES

(Default Values are all 'false')

CHGIND	read indicator changes in ENCDSG routine
KEEPCL	keep constituent list structure
NOPURG	do not do any purging
PRTSW	print each line on OUT6 as it is read
SAME	read more than one logical file from a physical file
TRACE	print TRMPNT and NAMEX arrays when writing binary file
TRADSG	trace ADDSEG routine
TRGIND	read optional data in ENCDSG routine
TRNS	trace NEWSEG routine
TRNS2	trace NEWSEG routine
TRSENT	print sentence numbers
TRWORD	print word number
TR1	trace switch, depends on routine
TR2	trace switch, depends on routine
TR3	trace switch, depends on routine
TR4	trace switch, depends on routine
TR5	print record identification numbers
TR6	read optional data at the beginning of some routines

APPENDIX D

Attributes of a segment type record

APFCR	1	A list of pointers to decoding rules which have this segment type as their first constituent
APPCL	2	A list of pointers to partial constituent lists which are waiting for a segment of this type
AUCL	3	Specifies the number of characters in a segment of this type, if it is unique.
AMCL	4	Special information (gets its value from the number after a star on the right side of a decoding rule).
ARULES	8	A list of encoding rules which have this segment type on the left.
ATRLEV	9	The "trace level" specified (optionally) during the processing of rules.
ANMS	10	The EBCDIC representation of the name of this segment type.

APPENDIX D

Attributes of a segment type record

APFCR	1	A list of pointers to decoding rules which have this segment type as their first constituent
APPCL	2	A list of pointers to partial constituent lists which are waiting for a segment of this type
AUCL	3	Specifies the number of characters in a segment of this type, if it is unique.
AMCL	4	Special information (gets its value from the number after a star on the right side of a decoding rule).
ARULES	8	A list of encoding rules which have this segment type on the left.
ATRLEV	9	The "trace level" specified (optionally) during the processing of rules.
ANMS	10	The EBCDIC representation of the name of this segment type.

APPENDIX E

SYMBOL CODES

TSCODE	SYMBOL	MEANING
1		Numeric Constant
2	" "	EBCDIC Character String
3	' '	Named Record Name
4		Indicator Name
5		Literal Record Name
6		Routine Name
7		Literal Attribute Name
8	,	Element Separator
9	¬	Logical Not
10	¢, %	Copy
11	=	Assignment
12	+	Addition
13	-	Subtraction, Deletion
14	*	Multiplication (also star)
15	/	Division
16	. .	Comparison
17	$\$_0$	"In the set" Test
18	$\$_1$	"In the set" Value
19	@	Attribute
20	(	Left Parenthesis
21	)	Right Parenthesis
22	|	Or

23	$\overset{=}{,}$	Special Symbol
24	<	Less Than
25	>	Greater Than

APPENDIX F

INSTRUCTION CODES

Value	Meaning
0	No operation (Separates constituents of a rule)
1,2	Set record pointer to attribute value
3,4	Set attribute number, for indirect specification
5,6	Get value from an attribute
7,8	Set an attribute to a value
9,10	Set an indicator to a value
11,12	Turn an indicator on
13,14	Test for indicator on
15,16	Test for indicator off
17,18	Turn an indicator off
19,20	Get a value from an indicator
21	Set indicator from a named record
22	Test for indicator on in a named record
23	Test for indicator off in a named record
24	Set record pointer to named record
25	Get a pointer to named record for value
26	Set attribute 1 to point to a named record
27	Test for attribute 1 pointing to a particular named record
28	Test for attribute 1 not pointing to a particular named record
29	Get record pointer for value
30	Set record pointer for literal record name
31	Get value from numeric constant

32	Get value from EBCDIC character string
33	Copy attribute
34	Test for presence of attribute
35	Test for absence of attribute
36	Test for set membership
37	Test for lack of set membership
38	Get copy of record for value
39	Make segment be a copy of specified record
40	Erase an attribute
41	Subtract
42	Add
43	Divide
44	Multiply
45	Test for specified comparison
46	Set attribute number, for indirect specification
47	Make segment be a copy (automatic)
48	Call a specified routine
49	Or
50	Find attribute value in a set
51	Set contextual constituent variable
52	Use same segment record instead of making an automatic copy
53 to 63	Not used
255	No operation (Signifies the ending of the left part or right part of a rule)

LIST OF REFERENCES

1. Chomsky, Noam, Aspects of the Theory of Syntax, MIT Press, Cambridge, Mass., 1965.

2. Lamb, Sydney M., Outline of Stratificational Grammar, Georgetown University Press, Washington, D.C., 1966.

3. Heidorn, George E., Natural Language Inputs to a Simulation Programming System - An Introduction, Naval Postgraduate School Technical Report, No. NPS-55HD71121A, December 1971.

4. Heidorn, George E., Natural Language Inputs to a Simulation Programming System, Naval Postgraduate School Technical Report, Forthcoming.

5. Hansen, Richard C., Ges: A Data-Structure-to-GPSS Encoding System, M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1970.

6. McGee, Robert T., The Translation of Data Structure Representations of Simple Queuing Problems into GPSS Programs and English Text, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1971.

7. Baker, Eldon S., Question-Answer Inputs to a Simulation Program Generating System, M.S. Thesis, U. S. Naval Postgraduate School, June 1971.

8. Hemphill, Frederick H., Computer Verification of the Completness of a Simulation Problem Description by Natural Language Interaction, M.S. Thesis, U.S. Naval Postgraduate School, December 1971.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Assistant Professor George E. Heidorn, Code 55HD Department of Operations Research Naval Postgraduate School Monterey, California 93940	5
4.	Captain Alfred H. Mossler 394-A, Ricketts Road, Monterey, California 93940	1

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)

Naval Postgraduate School
Monterey, California 93940

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

3 REPORT TITLE

Improving the Efficiency of a Natural Language Processor

4 DESCRIPTIVE NOTES (Type of report and, inclusive dates)

Master's Thesis; June 1972

5 AUTHOR(S) (First name, middle initial, last name)

Alfred H. Mossler

6. REPORT DATE June 1972	7a. TOTAL NO. OF PAGES 64	7b. NO. OF REFS 8

8a. CONTRACT OR GRANT NO.

b. PROJECT NO.

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10 DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

Naval Postgraduate School
Monterey, California 93940

13. ABSTRACT

NLP is a processor being developed at the Naval Postgraduate School for research in natural language man-machine communication. With this system text can be translated into an entity-attribute-value information structure, and such a structure can be translated into text. These two processes, called decoding and encoding, respectively, are specified by writing "rules" in a language designed for this system.

This thesis reports on a scheme for storing these rules in the computer in a compact fashion, and describes the related routines. The savings in core storage and CPU time achieved by using this scheme are given for a particular application of NLP.

Security Classification

KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Artificial Intelligence						
Computational Linguistics						
Mechanical Translation						
Natural Language Processing						

Thesis 141383
M842 Mossler
c.1 Improving the efficiency of a natural language processor.